Algoritmi e strutture dati

Koci Erik

March 13, 2021

1 Complessità algoritmi

1. **Costo:** si riferisce al costo di un singolo algoritmo
2. **Complessità:** si riferisce a più risoluzioni di un algoritmo

il costo di un blocco **if-then-else** è $O(max\{f(n), g(n), h(n)\})$ cioè $\mathbf{O(1)}$.

1.1 Ordini di grandezza

1. $\Theta(f(n))$ se **cresce tanto quanto f**
2. $O(f(n))$ **se la crescita è minore o uguale a f**
3. $\Omega(f(n))$ **se la crescita è maggiore o uguale a f**

1.2 Esercizi

- $1325n^2 + 12n + 1 = \theta(n^3)$ FALSO
- $76n^3 == (n^3)$ VERO
- $n^2 logn = O(n^2)$ FALSO
- $3^N = O(2^N)$ FALSO
- $1^n = O(2^{\frac{n}{2}})$ FALSO
- $2^N + 100 = O(2^N)$ VERO

- $n = O(nlogn)$ VERO
- $n^2 = (nlogn$ FALSO
- $log(n^2) = \Theta(logn)$ VERO
- $(n+1)/2 = \Theta(n)$ VERO
- $\frac{(n+1)*n}{2} = \Theta(n^2)$ VERO

1.3 Analisi casi

1. **Caso pessimo**
$$T\textsc{worst}(n) = maxT(I)$$
2. **Caso ottimo**
$$T\textsc{best}(n) = minT(I)$$
3. **Caso medio**
$$T\textsc{avg}(n) = \sum T(I)P(I)$$

1.4 Algoritmi ordinamento

Selection sort: scorre tutti gli elementi degli array e si cerca il valore più piccolo scambiando i due valori. Il costo è lineare con il numero di elementi da considerare:
$$T(n) = \Theta(n^2)$$
il costo è $\Theta(n^2)$ perchè è presente una funzione min che ogni volta controlla se il numero è il minore. Le chiamate a **min** contribuiscono a n^2 mentre il resto combacia con n cioè $n^2 + n$; n viene assorbito.

Ricerca binaria (ricorsiva): per utilizzare questo algoritmo devo avere un array ordinato. Cerco il valore andando a verificare sempre nella metà dove mi aspetto che sia presente.

$$T(n) = 1 \ \ se\ n = 0$$

$$T(n) = T(n/2) + 1$$

equazione di ricorrenza: ci aiuta a calcolare il costo analizzando una singola ricorsione.

1.5 metodo dell'iterazione:

consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione.

$$T(n) = c_1 + c_2 * log(n) = \Theta(log(n))$$

E' presente il **logaritmo** perchè ogni volta devo **dimezzare** il tutto in base al numero di elementi. c_1 perchè devo eseguire le istruzioni la prima volta.

dimostrare che $T(n) = O(n)$

$$T(n = 1)\ n == 1$$

$$T(T([n/2]) + n\ \ n > 1$$

1.6 Metodo della sostituzione:

consiste facendo una dimostrazione per induzione. quindi parto dal valore base che è n (esempio 1) e dimostriamo che vale anche per un n più grande.

caso base:

$$T(1) = 1 \leq cx1$$

induzione:

$$T(n) = T([n/2]) + n$$
$$\leq c[n/2] + n \quad (ipotesi\, induttiva)$$
$$\leq cn/2 + n = \frac{cn + 2n}{2} = (c/2 + 1)n \leq cn$$

1.7 Master Theorem:

Si consideri la seguente equazione di ricorrenza:

$$T(n) = d \;\; se \; n = 1$$
$$T(n) = aT(n/b) + cn^{\beta} \;\; se \; n > 1$$

e sia:

$$\alpha = \frac{log(a)}{log(b)}$$

a numero di chiamate ricorsive
b mi dice come partiziono il mio input
questi due valori mi danno α.
β mi dice l'esponente che avevo.

L'equazione di ricorrenza ha la seguente soluzione:

1. $T(n) = \Theta(n^{\alpha})$ se $\alpha > \beta$
2. $T(n) = \Theta(n^{\alpha} * log(n))$ se $\alpha = \beta$
3. $T(n) = \Theta(n^{\beta})$ se $\alpha < \beta$

Il teorema fondamentale **non** si può applicare ad algoritmi ricorsivi che non effettuano **partizioni bilanciate**.
ad esempio non può essere applicato nella risoluzione di fibonacci ricorsivo.

Esempio:

$$T(n) = 1 \, n <= 1$$
$$T(n1) + T(n2) + 1 \, n > 2$$

Se le **partizioni sono bilanciate** conviene utilizzare il **Master Theorem**.

partizione bilanciate: quando facciamo chiamate ricorsive prendo il mio input suddividendolo in parti n/b. Fibonacci non è bilancianto perche abbiamo due chiamate ricorsive diverse.

L'analisi ammortizzata: studia il costo medio di una sequenza di operazioni.

Sia **T(n, k)** il tempo totale richiesto da un algoritmo, nel caso pessimo, per effettuare k operazioni su istanze di lunghezza n. Definiamo il **costo ammortizzato** su una sequenza di k operazioni come:

$$T_\alpha(n) = \frac{T(n,k)}{k}$$

2 Scelta degli algoritmi

A seconda delle operazioni da eseguire è necessario adattare diverse tecniche di implimentazione di un algoritmo.

2.0.1 Implementazione su un vettore ordinato

Questo tipo di ricerca ha un costo computazionale basso nel caso in cui volessimo **ricercare degli elementi**. Costi computazionali:

- Ricerca $O(log\, n)$
- Inserimento $O(n)$
- Eliminazione $O(n)$

2.0.2 Implementazione su liste concatenate non ordinate

Questo implementazione converrebbe utilizzarla nel caso in cui volessimo **aggiungere o eliminare degli elementi**. Costi computazionali:

- Ricerca $O(n)$
- Inserimento $O(1)$
- Eliminazione $O(n)$

2.1 Algoritmi di visita degli alberi

Esistono due tipologie di visita:

- In profondità (pre-ordine, in-rodine, post-ordine)
- In ampiezza

Nella visita **pre-ordine** si parte visitando il nodo della radice per poi passare a visitare tutto il nodo sinistro, risalendo poi andando verso destra.

Nella visita **in-ordine** si parte a visitare dal ramo più in basso a sinistra risalendo per poi andare verso destra.

Nella visita **post-ordine** vengono prima visitati i nodi più in profondità partendo sempre da sinistra verso destra per poi risalire.

Nella visita per **ampiezza** si analizza l'albero a livelli, partendo dalla radice.